A Modeling and Formal Approach for the Precise Specification of Security Patterns

Brahim Hamid, Christian Percebois

A Modeling and Formal Approach for the Precise Specification of Security Patterns

Brahim Hamid and Christian Percebois

IRIT, University of Toulouse
118 Route de Narbonne, 31062 Toulouse Cedex 9, France
{hamid,percebois}@irit.fr

Abstract. Non-functional requirements such as Security and Dependability (S &D) become more important as well as more difficult to achieve. In fact, the integration of security features requires the availability of both application domain specific knowledge and security expertise at the same time. Hence, capturing and providing this expertise by the way of *security patterns* can support the integration of S&D features by design to foster reuse during the process of software system development.

The solution envisaged here is based on combining metamodeling techniques and formal methods to represent security pattern at two levels of abstraction fostering reuse during the process of pattern development and during the process of pattern-based development. The contribution of this work is twofold: (1) An improvement of our previous pattern modeling language for representing security pattern in the form of a subsystem providing appropriate interfaces and targeting security properties, (2) Formal specification and validation of pattern properties, using the interactive Isabelle/HOL proof assistant. The resulting validation artifacts may mainly complete the definitions, and provide semantics for the interfaces and the properties in the context of S&D. As a result, validated patterns will be used as bricks to build applications through a Model-Driven engineering approach.

Keywords: Pattern, Metamodel, Domain, Formalization, Model-Driven engineering, Security.

1 Introduction

Recent times have seen a paradigm shift in terms of design by combining multiple software engineering paradigms, namely, Model-Driven Engineering (MDE) [20] and formal methods [25]. Such a paradigm shift is changing the way systems are developed nowadays, reducing development time significantly. Embedded systems [26] are a case where a range of development approaches have been proposed. The most popular are those using models as main artifacts to be constructed and maintained. In these approaches, software development consists of model specification and transformations. In addition to MDE, pattern-based development has gained more attention recently in software engineering by addressing new challenges that were not targeted in the past. In fact, they are applied in modern software architecture for distributed systems including middlewares, real-time embedded systems, and recently in security and dependability engineering.

Unfortunately, most of security patterns are expressed as informal indications on how to solve some security problems, using template like traditional patterns [1,6]. These patterns do not include sufficient semantic descriptions, including those of security and dependability concepts, for automated processing within a tool-supported development and to extend their use. Furthermore, due to manual pattern implementation, the problem of incorrect implementation (the most important source of security problems) remains unsolved. For that, model driven software engineering can provide a solid basis for formulating design patterns that can incorporate security aspects and for offering these patterns at several levels of abstraction.

In this paper, we leverage on this idea to propose a new framework for the specification and the validation of security patterns intended for systems with stringent security requirements. Reaching this target requires to get a common representation of such a modeling environment for several domains and the ability to customize them for a specific domain. Here we propose two additional and complementary types of representations: a semi-formal representation through metamodeling techniques and a rigorous formal representation through interactive theorem proving approaches. Regarding the comparison with documentation template [6] (informal representation) used to represent security patterns, our proposition is based on it. However, we keep the template elements in the form of attributes and we deeply refine them by the definition of new concepts in order to fit with security engineering needs. In this vision, a security or (in general an S&D) pattern is a subsystem exposing pattern functionalities through interfaces and targeting security properties. As it follows the MDE paradigm for system's design, using patterns on different levels of abstraction, it allows for integration into the system's design process, hence supports this process. To this end, the proposed representation takes into account the simplification and the enhancement of such activities, namely: selection/search, based on the classified properties, and integration, based on a high level description of interfaces.

The rest of this paper is organized as follows. An overview of the modeling approach we proposed including a set of definitions is presented in Section 2. Then, Section 3 presents the pattern modeling language and illustrates the pattern modeling process in practice. Section 4 presents the validation process through the example of the secure communication pattern. In Section 5, we review most related works addressing pattern specification and validation. Finally, Section 6 concludes this paper with a short discussion on future works. An appendix is added with the Isabelle/HOL code of our experiment.

2 The Nature of Patterns within PBSE (Pattern-Based System and software Engineering)

Usually, pattern design artifacts are provided as a library of models (subsystems) and as a system of patterns (framework) in the more elaborated approaches. However, there are still lacks in existing modeling languages and/or formalisms dedicated to model these design artifacts and the way how to reuse them in software development automation. The supporting research activities examine three distinct challenges: mining - discovering patterns from existing systems, hatching - selection of the appropriate pattern and application - effective use during the system development process.

In our work, we study only the two last challenges, targeting the (i) development of an extensible design language for modeling patterns for secure and dependable distributed embedded systems and (ii) a methodology to improve existing development processes using patterns. The language has to capture the core elements of the pattern to help its (a) precise specification, (b) selection and (c) integration. Supporting research tackles the presented challenges includes domain patterns, pattern languages and recently formalisms and modeling languages to foster their application in practice. Here, we propose a novel approach to improve the security pattern representation by the combination of semi-formal semantics and rigorous formal semantics of some of its concepts.

2.1 Motivational Example: Secure Communication Pattern (SCP)

As example of a common and a widely used patterns we choose the Secure Communication Pattern referred to in the following as SCP. Messages passing across any public network can be intercepted. The problem is how to ensure that data being across this system is secure in transit. In other words, how to guarantee data authenticity. This is one of the goal to use the SCP.

However, those SCP are slightly different with regard to the application domain. For instance, a system domain has its own mechanisms and means to serve the implementation of this pattern using a set of protocols ranging from SSL, TLS, Kerberos, IPSec, SSH, WS-Security and so on. In summary, they are similar in the goal, but different in the implementation issues. So, the motivation is to handle the modeling of security patterns by following abstraction. In the following, we propose to use SSL mechanisms [1] to specialize the implementation of the SCP.

The establishment of a secure channel is composed of two phases. First, the client informs the server of the cryptographic algorithms it can handle. The actual choice is always made by the server, which reports its choice back to client. In the second phase, authentication takes place. The server is required to authenticate itself. It sends a certificate containing its public key signed by a certification authority CA to the client. In order to authenticate the client by the server, the client has to send a certificate to the server as well. The client generates a random number that will be used by both sides for building a session key, and sends this number to the server, encrypted with the server's public key. In the case of client required authentication, the client signs the number with its private key. At that point, the sever can verify the identity of the client, after which the secure channel can be established.

2.2 Definitions and Concepts

In [6], a design pattern abstracts the key artifacts of a common design structure that make it useful for creating a reusable object-oriented design. Several generalizations on this basis to describe software design patterns in general are proposed in literature. We quote the following definition of security patterns from [21]:

[1] SSL or its update named TLS proposed in RFC 2246.

Definition 1 (Security Pattern). *A security pattern describes a particular recurring security problem that arises in specific contexts and presents a well-proven generic scheme for its solution.*

Security patterns are not only defined from a platform independent viewpoint (i.e. they are independent from the implementation), they are also expressed in a consistent way with domain specific models. Consequently, they will be much easier to understand and validate by application designers in a specific area. To capture this vision, we introduced the concept of *domain view*. Particularly a security pattern at domain independent level exhibits an abstract solution without specific knowledge on how the solution is implemented with regard to the application domain.

Definition 2 (Domain). *A domain is a field or a scope of knowledge or activity that is characterized by the concerns, methods, mechanisms, ... employed in the development of a system. The actual clustering into domains depends on the given group/community implementing the target methodology.*

In our context, a domain represents all the knowledge including protocols, processes, methods, techniques, practices, OS, HW systems, measurement and certification related to the specific domain. With regard to the artifacts used in the system under development, we will identify the first classes of the domain to specialize such artifacts. For instance, the specification of a pattern at domain independent point of view is based on the software design constructs. The specialization of such a pattern for a domain uses a domain protocol to implement the pattern solution (see example of secure communication pattern given in Section 3.2.1 and Section 3.2.2).

The objective is to reuse the domain independent model security patterns for several industrial application domain sectors and also let them be able to customize those domain independent patterns with their domain knowledge and/or requirements to produce their own domain specific artifacts. Thus, the 'how' to support these concepts should be captured in the specification languages.

3 Pattern Modeling Process

We now present an overview of our pattern specification process. Along this description, we will give the main keys to understand why our process is based on a general and a constructive approach. We begin with our pattern specification metamodel.

3.1 Pattern Specification Metamodel (SEPM)

To foster reuse of patterns in the development of critical systems with S&D requirements, we are building on a metamodel for representing security pattern in the form of a subsystem providing appropriate interfaces and targeting security properties to enforce the S&D system requirements. Interfaces will be used to exhibit pattern's functionality in order to manage its application. In addition, interfaces support interactions with security primitives and protocols for the specialization for a specific application domain. The principal classes of the System and Software Engineering Pattern Metamodel (SEPM) [11] are described with Ecore notations in Figure 1. Their meanings are more detailed in the following paragraphs.

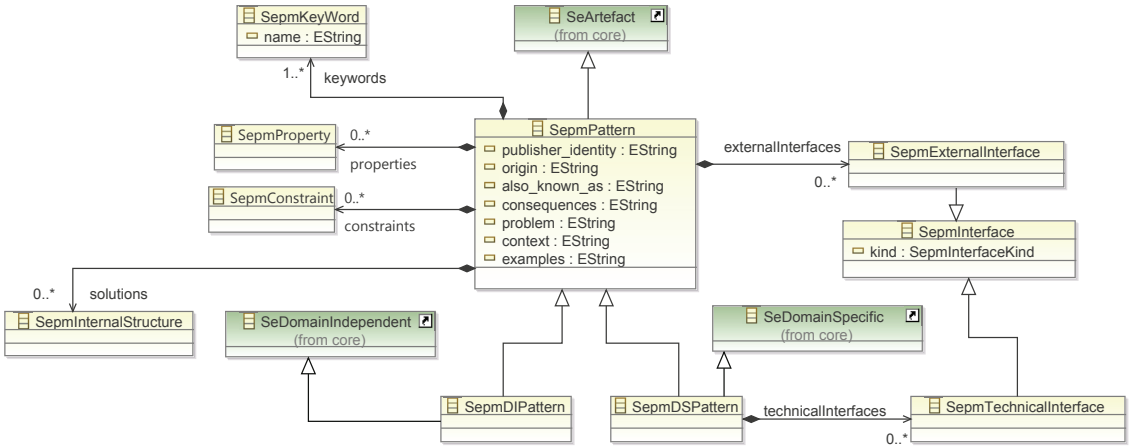

Fig. 1. The SEPM metamodel - Overview

- *SepmPattern.* This block represents a security pattern as a subsystem describing a solution for a security particular recurring design problem that arises in specific design context. A *SepmPattern* defines its behavior in terms of provided and required interfaces. Larger pieces of a system's functionality may be assembled by reusing patterns as parts in an encompassing pattern or assembly of patterns, and wiring together their required and provided interfaces. A *SepmPattern* may be manifested by one or more artifacts.
- *SepmDIPattern.* It is an *SepmPattern* denoting some abstract representation of a security pattern at domain independent level. This is the key entry artifact to model patterns at domain independent level (DIPM).
- *Interface.* A *SepmPattern* interacts with its environment with *Interfaces* which are composed of *Operations.* A *SepmPattern* owns provided and required interfaces. A provided interface highlights the services exposed to the environment. A required interface corresponds to services needed by the pattern to work properly. We consider two kinds of interface:
  - *External interface.* Allows implementing interaction with regard to the integration of a pattern into an application model or to compose patterns.
  - *Technical interface.* Allows implementing interaction with security primitives and protocols, such as encryption, and specialization for specific underlying software and/or hardware platforms, mainly during the deployment activity. Please note, a *SepmDIPattern* does not have *TechnicalInterfaces*.
- *Property.* Is a particular characteristic of a pattern related to the concern it is dealing with and dedicated to capture its intent in a certain way. For instance, security properties. Each property of a pattern will be validated at the time of the pattern validating process and the assumptions used, will be compiled as a set of constraints which will have to be satisfied by the domain application.
- *SepmDSPattern.* Is a refinement of a *SepmDIPattern*. It is used to build a pattern at at domain specific level (DSPM). Furthermore a *SepmDSPattern* has *Technical Interfaces* in order to interact with the platform. This is the key entry artifact to model pattern at DSPM.

3.2 Specification Process

We propose an iterative specification process consisting of the following phases: (1) the specification of the pattern at domain independent level (DIPM), (2) the refinement of

DIPM pattern to specify one of its representations at domain specific level (DSPM). These two levels of the secure communication pattern presented in Section 2.1 are illustrated. For the sake of clarity, many functions and artifacts of this pattern have been omitted. We only detail the properties and interfaces that we need to describe the validation process. Note that the artifacts representing the formalization and the proof are detailed in Section 4.

The first step in our specification process is the understanding of the pattern's informal representation. The target representation is the SEPM metamodel, simplified in Figure 1 for the purpose of our study (semi-formal and formal representation of security pattern). The informal description given in Section 2.1 reflects our understanding from the representation of secure communication pattern given in literature [22]. At the DIPM level this description reveals the following elements: interfaces of type *SepmExternalInterface* and security properties of type *SepmProperties*. At the DSPM level, the description reveals the following elements: interfaces of type *SepmExternalInterface* and *SepmTechnicalInterface* and security properties of type *SepmProperties*. The description with varying levels of abstraction is managed by inheritance. Once there is a good understanding of the pattern informal representation structure, the pattern can be specified using the SEPM metamodel. The first step is to create an instance of *SepmPattern*. The instance is given a set of attributes representing the pattern. In our example, an instance of *SepmPattern* is created and named 'SecureCommPattern'.

Once the basic pattern subsystem has been specified, interfaces are added to expose some of the pattern's functionalities. For each such interface, an instance of *SepmExternaInterface* is added to the pattern's interfaces collection. The next step after creating interfaces is the creation of properties instances. An instance is created in the pattern's properties collection to specify every identified security property. A property is given a name and an expression on external interfaces in a property language. A complete specification of this pattern at both DIPM and DSPM levels is described below.

3.2.1 Domain Independent Pattern Model (DIPM). At DIPM level, the security pattern subsystem and its related elements are created by inheritance. In our example, the DIPM of the secure communication pattern consists of:

- *External Interfaces.* The secure communication pattern exposes its functionalities through an external interface offering the following function calls:
  - $Send(P, Q, ch(P, Q), m)$,
  - $Receive(P, Q, ch(P, Q), m)$, with $P, Q \in \{C, S\}$, $ch(C, S) = ch(S, C)$ denoting the communication channel of client and server, and m a message.
- *Properties.* At this level, we specify the security property: "authenticity of sender and receiver". We denote this property as $auth(Send(C, S, ch(C, S), m), Receive(S, C, ch(C, S), m), S)$ where the server S invokes the primitives $Send(C, S, ch(C, S), m)$ and $Receive(S, C, ch(C, S), m)$.

3.2.2 Domain Specific Pattern Model (DSPM). At DSPM level, the security pattern and some of its related elements are also created by inheritance. Once a SEPMDSPATTERN is created, every pattern external interface is identified and modeled as a refinement of the SEPMEXTERNALINTERFACE in the pattern's interfaces collection. Then,

each of the pattern's technical interface is identified and modeled by an instance of *SepmTechnicalInterface* in the pattern's interfaces collection. The next step is the specification of properties. Each property is represented by an instance of *SepmProperty* in the pattern's properties collection. A property is given a name and an expression on external and technical interfaces in a property language.

For instance, when using SSL as a mechanism related to the application domain to refine the secure communication pattern at DSPM, we manage the following artifacts:

- *External Interfaces.* The DS external interface, a refinement of the DI external interface, can be specified as follows:
  - $send(C, S, mac_C(m), m)$: The client C sends m and the corresponding MAC (Message Authentication Code) to the server S.
  - $recv(S, C, mac_C(m), m)$: The server receives m and corresponding MAC.
- *Technical Interfaces.* The most important functions of the DS technical interface of the SSL pattern can be specified as follows:
  - $genRand(C, R_C), genRand(S, R_S)$: Client/server generate a random number.
  - $verifyCert()$: Client/server verify each others certificate and extract the respective public key.
  - $encrypt(C, pubKey_S, R_C)$: The client encrypts its random number using the server's public key.
  - $sign(C, \ldots)$: The client signs the SSL handshake messages.
  - $verifySig(S, \ldots)$: The server verifies the client's signature.
  - $genMac(C, macKey_C, m, mac_C(m))$: The client generates the MAC for message m using its own SSL shared secret for MAC generation.
  - $verifyMac(S, macKey_C, m, mac_C(m))$: The server verifies, using its shared secret for MAC verification (i.e. the client's key for MAC generation), that the MAC for m is correct and thus originates from the client.
  - $send(), recv()$: Send and receive of the SSL messages by client and server, respectively.
- *Properties.* In addition to the refinement of the authenticity property identified in the DIPM, at this level we may identify some related resource properties, e.g. the size of the cryptographic key.

4 Pattern Validation Process

We propose to use the interactive Isabelle/HOL proof assistant [17] for formalizing security patterns. The formal specification and verification of a pattern is represented through an Isabelle theory. It consists of a set of types, functions and theorems. In some situations, intermediate lemmas have been introduced in order to simplify what we have to prove.

We have defined a set of transformation rules to map the pattern metamodel concepts onto those of Isabelle/HoL. In our case, we focus on interfaces and properties. These interfaces provide functionalities through function calls that are encoded as actions whose parameters are agents and data. On the other side, pattern properties denote some expected outcomes over these actions. A property is given a name and an expression on agents and actions. Sometimes a property may refer to another property.

Each concept involved in the modeling of interfaces and properties is translated in Isabelle as an abbreviation, i.e. a new name for an existing construction, using data-types, records and definitions. For instance, the record *action* and the abbreviation *createAction* allow to create the *Send* and *Receive* actions. Following this definitional approach, we build the pattern specification from the bottom.

The designer can then extend this theory by verifying properties using the Isabelle scripting language for writing proofs using tactics. The goal now is to identify the set of assumptions required to prove the properties. Then, the proof consists to find a scheduling of valid steps using Isabelle's tactics such as applying a simplification considering that each step corresponds to a sub-goal to resolve (*command apply*). Our proof only uses simplification rules which consist in rewriting specified equations from left to right in the current goal (*command simp*).

Correctness of the proof is guaranteed by construction from the first goal to resolve (a lemma or a theorem) until the message "no sub-goals" is produced by the framework which confirms that the proof is finished (*command done*). For instance, to prove the authenticity property expressed as the goal $lemma\ authSendReceive$: $''auth\ Send\ Receive\ server''$ with respect to the definition *auth*, each subgoal *auth*, *Send*, *Receive*, ... is rewritten according to its definition.

In the following, we apply the formalization and the verification processes to both DIPM and DSPM levels, and then we present the poof that the DIPM is an abstraction of the DSPM. The definitions introduced below are extracted from the code of our experiment. Definitions mainly concern the refinement process of SCP from the DIPM model to the DSPM model, using the SSL mechanism for the application domain.

4.1 Pattern Formalization

We first introduce the main concepts of authenticity, precedence and trust that are relevant for SCP. We call a particular action a authentic for an agent P after a sequence of actions if in all sequences that P considers to have possibly happened a must have happened. This is modeled by the $auth(a, b, P)$ predicate which denotes that whenever a particular action b has happened, it must be authentic for agent P that action a has happened as well. In the same way, $precede(a, b)$ holds if all action sequences in the system's behavior that contain an action b also contain an action a. Finally, $trust(P, prop)$ means that P trusts a property $prop$ to hold in the system if the property holds in the agent's conception of the system. The following gives types for these predicates in Isabelle.

$$auth ::'' action => action => agent => bool''$$
$$precede ::'' action => action => bool''$$
$$trust ::'' agent => property => bool''$$

Before encoding the authenticity relation in Isabelle/HOL, one has to define what are an action, an agent and a property. All three are records i.e. tuples, with a name attached to each component. They will be independently introduced when defining the DIPM and DSPM pattern models respectively in Figure 2 and Figure 3.

```
1: record action = agent1 :: "agentType"
                agent2 :: "agentType"
                channel :: "functorType"
                message :: "messageType"
2: definition createAction ::"agentType==>agentType==>functorType==>messageType==>
action" where "createAction a1 a2 ca me==agent1=a1,agent2=a2,channel=ca,message=me"
3: definition Send :: "action" where "Send == createAction C S ch m"
4: definition Receive :: "action" where "Receive == createAction S C ch m"
5: record property = action1 :: "action"
                action2 :: "action"
6: definition createProperty :: "action==>action==>property" where
"createProperty a1 a2 == action1=a1, action2=a2"
7: definition precede :: "action==>action==>property" where
"precede a1 a2 == createProperty a1 a2"
8: definition precedeSendReceive :: "property" where
"precedeSendReceive == precede Send Receive"
9: record agent = name :: "agentType"
                actions :: "action set"
                properties :: "property set"
10: definition createAgent :: "agentType==>(action set)==>(property
set)==>agent" where "createAgent a as ps == name=a, actions=as, properties=ps"
11: definition client :: "agent" where "client == createAgent C {Send} {}"
12: definition server :: "agent" where
"server == createAgent S {Send, Receive} {precedeSendReceive}"
13: definition auth :: "action==>action==>agent==>bool" where
"auth a1 a2 a == a1 IN (actions a) AND a2 IN (actions a)"
14: definition trust :: "agent==>property==>bool" where
"trust a p ==  p IN (properties a)"
15: definition authIfTrust :: "action==>action==>agent==>bool" where
"authIfTrust a1 a2 a == trust a (precede a1 a2)−>auth a1 a2 a"
```

Fig. 2. The DIPM formalization of the Secure Communication Pattern

4.1.1 Domain Independent Pattern Model (DIPM). As previously defined in Section 3.2.1, the secure communication pattern exposes its functionalities through the two primitives $Send$ and $Receive$. Applying the previous definition $auth$, we have to define these two primitives at the DIPM level. Figure 2 highlights the corresponding formalization in Isabelle owing to the $createAction$ definition and the $Send$ and $Receive$ constants (lines 2-4). In the same way, an agent is introduced by the $createAgent$ definition and the subject constant nominates the active computational entity of the pattern (line 10). The modeled cooperating system is so composed of a set of agents (e.g. some clients and a server) and a set of actions (e.g. the $Send$ and $Receive$ interface actions introduced above). We consider that the set of all possible sequences of actions models the behavior of the system and that an agent's initial knowledge about the system consists of all traces the agent initially considers possible. An agent may assume for example that a message that was received must have been sent before, and this is the case for any validation of a security property holding in a system. In our example, the SCP's DIPM level states that a $Receive$ action from a client C using a specific channel ch is always preceded by the respective $Send$ generation action triggered by the server S (line 8).

4.1.2 Domain Specific Pattern Model (DSPM). The formal model corresponding to the DSPM introduced in Section 3.2.2, as the refinement of the DIPM using the SSL mechanism, contains the same set of agents, namely C and S. Figure 3 depicts

```
1: record actionMac = agent :: "agentType"
                        functor1 :: "functorType"
                        data :: "messageType"
                        functor2 :: "functorType"
2: definition createActionMac :: "agentType==>functorType==>messageType ==>
functorType ==> actionMac" where "createActionMac a f1 d f2 == (agent=a,
functor1=f1, data=d, functor2=f2)"
3: definition genMac :: "actionMac" where
"genMac == createActionMac C macKey m mac"
4: definition verifyMac :: "actionMac" where "verifyMac == createActionMac S
macKey m mac"
5: record propertyMac = action1 :: "actionMac"
                       action2 :: "actionMac"
6: definition createPropertyMac :: "actionMac==>actionMac==>propertyMac" where
"createPropertyMac a1 a2 == (action1=a1, action2=a2)"
7: definition precede :: "actionMac ==>actionMac==>propertyMac" where
"precede a1 a2 == createPropertyMac a1 a2"
8: definition precedeGenMacVerifyMac :: "propertyMac" where
"precedeGenMacVerifyMac == precede genMac verifyMac"
9: record agent =  name :: "agentType"
              actionsMac :: "actionMac set"
              actionsRandom :: "actionRandom set"
              actionsKey :: "actionKey set"
              propertiesMac :: "propertyMac set"
              propertiesRandom :: "propertyRandom set"
              propertiesKey :: "propertyKey set"
10: definition createAgent :: "agentType==>(actionMac set)==>(actionRandom set)
==>(actionKey set)==>(propertyMac set)==>(propertyRandom set)==>(propertyKey
set)==>agent" where "createAgent a asm asr ask psm psr psk == (name=a, actionsMac=asm,
actionsRandom=asr, actionsKey=ask, propertiesMac=psm, propertiesRandom=psr,
propertiesKey=psk)"
11: definition server :: "agent" where "server == createAgent S {genMac,verifyMac}
{genRand} {privKeyS, privKeyCA} {precedeGenMacVerifyMac} {notPrecedeGenRandGenRand}
{confPrivKeyS, confPrivKeyCA}"
12: definition trustMac :: "agent==>propertyMac==>  bool" where
"trust a p ==  p IN (propertiesMac a)"
```

Fig. 3. The DSPM formalization of the Secure Communication Pattern

some of these DSPM artifacts using the Isabelle definitions. Its actions correspond to the external and technical DS interface function calls. They can be considered as a refinement of the actions of the DIPM formal model. The code defines also a specific DSPM server S suitable for the SSL protocol (line 11). The security property that is provided by this SSL pattern and that corresponds to the trust property assumed to hold for the DIPM model is that the server trusts into the precedence of its own MAC verification action by the MAC generation action of the client (line 12).

4.2 Pattern Validation

Now we present the verification of the secure communication pattern at both DIPM and DSPM levels.

4.2.1 Domain Independent Pattern Model (DIPM). The DIPM formal model of Figure 2 defines the SCP's collaborative agents of the pattern refined at Figure 3 using the SSL mechanism. The former figure also introduces the $Send$ and $Receive$ actions corresponding to the external DI interface (lines 3 and 4). We further assume that each agent can only see its own actions. According to Section 3.2.1, the required authenticity

property is expressed as: $lemma\ authSendReceive: \ ''auth\ Send\ Receive\ server''$ (P-DI)

In order to prove this constraint between $Send$ and $Receive$ messages, we use the result of [5] which states that the properties $trust(P, precede(a, b))$ and $auth(b, c, P)$ imply the property $auth(a, c, P)$. Setting $b = c$ and instantiating a, b, P with the concrete $Send$ and $Receive$ actions and the server S of property P-DI, we conclude that P-DI holds if the following properties (assumptions) hold:
- $lemma\ trustSendReceive: \ ''trust\ server\ precedeSendReceive''$ (A-DI)
- $lemma\ authReceiveReceive: \ ''auth\ Receive\ Receive\ server''$
We may assume that the latter property holds because any action identified by the server is authentic. Hence, this concludes our proof with respect to the DIPM model the assumption (A-DI) must be assumed to hold.

We emulate these constraints in Isabelle by the theorem:
$theorem\ authIfTrustSendReceive: \ ''authIfTrust\ Send\ Receive\ server''$
which terminates the P-DI proof of the DIPM model assuming the assumption A-DI. For simplification purpose, we consider that trust of the server into the precedence of a corresponding $Receive$ action by a client $Send$ action is given by the set membership relation of the same action to the set of actions of mapped agents. Recall that Figure 2 introduces the definitions which encodes the precedence, auth and trust properties between a $Send$ action and a $Receive$ action for the couple client and server (lines 8, 13, 14 and 15).

The fact that no more DIPM definition can be applied shows that we now have to consider the DSPM level, i.e. we have to find and validate the SSL implementation that provides an equivalent property. This will be discussed in the next paragraph.

4.2.2 Domain Specific Pattern Model (DSPM). The security property that is provided by this SSL pattern and that corresponds to the trust property assumed to hold for the DIPM model is that the server trusts into the precedence of its own signature verification action by the signature generation action of the client:

$lemma\ trust: \ ''trust\ server\ precedeGenMacVerifyMac''$ (P-DS)

In order to prove P-DS, we isolate assumptions provided by the SSL protocol as a random number is only generated once, the server trusts into the confidentiality of its own private RSA key and into the confidentiality of the certificate authority's private key. These statements captures the semantics of an RSA encryption and allows to conclude that S trusts in the confidentiality of the shared secrets derived from the SSL handshake and yields that indeed property P-DS holds. For more details, the complete proof was introduced in [11]. In our case, we introduce the $genMac$ and $verifyMac$ actions and their temporal dependency $precedeGenMacVerifyMac$, as illustrated by Figure 3 (lines 3, 4 and 8). We also defined the trust predicate for the DSPM server (line 12). This predicate will be used by the refinement process from the DIPM model to the DSPM model (line 1 of Figure 4).

We synthesize the P-DS proof by introducing the $genMac$ and $verifyMac$ actions and their temporal dependency $precedeGenMacVerifyMac$, as illustrated by Figure 3 (lines 3, 4 and 8). However we need what is trust for a DSPM server; this is

defined by the code of line 12. This predicate will be used by the refinement process from the DIPM model to the DSPM model (see line 1 of Figure 4).

4.3 Correspondence between DIPM and DSPM

Note that a system specification does not require a particular level of abstraction. Different formal models of the same system are partially ordered with respect to different levels of abstraction. Formally, abstractions can be mapped to action sequences of a finer abstraction level to action sequences of a more abstract level while respecting concatenation of actions.

Correspondence between the DIPM and DSPM formal models is assumed when proving that the property introduced at the DIPM model is transferred to the DPSM model. More precisely, this means that the DIPM model is an abstraction of the DSPM model. In particular, we must show that using a specific mechanism for verifying the property together with function calls of the specific domain is a specific case of proving the upper-level property.

We so have to map actions of the DSPM model onto the actions of the DIPM model by an appropriate homomorphism h and then prove that this homomorphism preserves *trust* in *precede*. In practice, h is required to preserve each operation or a pseudo-operation which summarizes the behavior of a set of operations. Figure 4 specifies h in Isabelle and the resulting $trustWithH$ theorem.

```
1:definition h :: "(DSPM.action * DIPM.action) list" where
"h == [(genMac, Send), (verifyMac, Receive)]"
2:definition buildProperty :: "(DSPM.action * DIPM.action) list==>DSPM.property"
where "buildProperty l == DSPM.property.action1=fst(nth l 0),
DSPM.property.action2=fst(nth l 1)"
3:definition applyH :: "(DSPM.action * DIPM.action) list==>DIPM.property" where
"applyH l ==DIPM.property.action1=snd (nth l 0), DIPM.property.action2=snd(nth l 1)"
4:definition buildAndApplyH :: "DSPM.agent==>DIPM.agent==>
(DSPM.action * DIPM.action) list==>bool" where
"buildAndApplyH a1 a2 l == DSPM.trust a1(buildProperty l)−>
DIPM.trust a2 (applyH l)"
5:theorem trustWithH : "buildAndApplyH DSPM.server DIPM.server h"
```

Fig. 4. Proving *trust* in *precede* for the Secure Communication Pattern

Since we assume property A-DI to hold for the DIPM model, all server $Receive$ actions are preceded by a client $Send$ action in the server's abstract initial knowledge. On the other hand, the server's concrete initial knowledge reflects the MAC mechanism, i.e. reflects that a $verifyMac$ action is always preceded by the respective $genMac$ action. This is assumed by the $buildProperty$ definition of Figure 4 (line 2) introduced as assumption and by the $precedeGenMacVerifyMac$ definition of Figure 3 (line 8).

Note that we consider P-DS as the premise of an implication where $DSPM.trust\ a1\ (buildProperty\ l)$ refers to $trust$ for the DSPM model of Figure 3, while $DIPM.trust\ a2\ (applyH\ l)$ refers to $trust$ for the DIPM counterpart of Figure 2. More generally, the DIPM and DSPM call or type prefixes of Figure 4 are related to each modeling. Thereby, setting up and proving the $trustWithH$ theorem (line 5) requires defining both a DIPM server and a DSPM server; these two servers have the same

functionality in practice. Based on these assumptions, the homomorphism h preserves trust into precedence and property A-DI transferred to the DSPM model is identical to property P-DS.

5 Related Works

Design patterns are a solution model to generic design problems, applicable in specific contexts. Supporting research tackles the presented challenges includes domain patterns, pattern languages and recently formalisms and modeling languages to foster their application in practice. To give an idea of the improvement achievable by using specific languages for the specification of patterns, we look at pattern formalization and modeling problems targeting the integration of the pattern specification and validation steps into a broader MDE process.

Several tentatives exist in the literature to deal with patterns for specific concerns [8,23]. They allow to solve very general problems that appear frequently as sub-tasks in the design of systems with security and dependability requirements. These elementary tasks include secure communication, fault tolerance, etc. The pattern specification consists of a service-based architectural design and deployment restrictions in form of UML deployment diagrams for the different architectural services.

To give an overview of the improvement achievable by using specific languages, we look at the pattern specification and formalization problems. UMLAUT [9] is an approach that aims to formally model design patterns by proposing extensions to the UML metamodel 1.3. They used OCL language to describe constraints (structural and behavioral) in the form of meta-collaboration diagrams. In the same way, RBML (Role-Based Metamodeling Language) [15] is able to capture various design perspectives of patterns such as static structure, interactions, and state-based behavior.

While many patterns for specific concern have been designed, still few works propose general techniques for patterns. For the first kind of approaches [6], design patterns are usually represented by diagrams with notations such as UML objects, annotated with textual descriptions and examples of code. There are some well-proven approaches [3] based on Gamma et al. However, this kind of technique does not allow to reach the high degree of pattern structure flexibility which is required to reach our target.

Formal specification has also been introduced in [16] in order to give rigorous reasoning of behavioral features of a design pattern in terms of high-level abstractions of communication. In this paper, the author considers an object-oriented formalism for reactive system (DisCo) [13] based on TLA (Temporal Logic of Actions) to express high-level abstractions of cooperation between objects involved in a design pattern. However, patterns are directly formalized at the pattern level including its classes, its relations and its actions, without defining a modeling language.

The work in [2] introduces a new specification template inspired on secure system development needs. The template is augmented with UML notations for the solution and with formal artifacts for the requirement properties. Another approach [7] provides a formal and visual language for specifying design patterns called LePUS. It defines a

pattern in an accurate and complete form of formula in Z, with a graphical representation. The framework promoted by LePUS is interesting but the degree of expressiveness proposed to design a pattern is too restrictive.

UMLsec [14] is an approach based on the integration of modeling security in UML and formal methods for object-oriented system development. UMLsec is defined in form of a UML profile and semantics to assist in the automated analysis of the UMLsec models with respect to security requirements. UMLsec and our approach are not in competition but they complement each other by providing different view points to the secure information system, mainly in applying security patterns for system security engineering.

Moreover, [12] used the concept of security problem frames as analysis patterns for security problems and associated solution approaches. The analysis activities using these patterns are described with a highlight of how the solution may be set, with a focus on the privacy requirement anonymity. For software architecture, [10] presented an evaluation of security patterns in the context of secure software architectures. The evaluation is based on the existing methods for secure software development, such as guidelines as well as on threat categories.

To summarize, in software engineering, design patterns are considered effective tools for the reuse of specific knowledge. However, a gap between the development of systems using patterns and the pattern information still exists. This becomes even more visible when dealing with specific concerns namely security and dependability properties for several application sectors such as presented recently in [4]. In an other point of view, we agree with the argumentations given in [24] to justify why the precise specification and formalization of a pattern by definition restricts its ”degree of freedom for the design”, and hence there is no success stories of works dealing with pattern formalization. This is not only related to security patterns. Note however, that this work does not address the validation activity which is an important issue in any design activity and more particularly in security engineering. We think that security is subject to rigorous and precise specification and the proposed literature (in our best knowledge) fails to meet these two objectives. To remedy these contradictory needs, we support the specifications of security patterns at two levels of abstractions, domain independent and domain specific, in both a semi-formal representation through metamodeling techniques and a rigorous formal representation through interactive theorem proving approach. This allows to support some variability of the pattern.

6 Conclusion and Future Work

A classical form of pattern is not sufficient to tame the complexity of safety critical systems – complexity occurs because of both the concerns and the domain management. To reach this objective and to foster reuse, we introduced the specification at domain independent and domain specific levels. The former exhibits an abstract solution without specific knowledge on how the solution is implemented with regard to the application domain. Following an MDE process, the domain independent model of patterns is then refined towards a domain specific level, taking into account domain artifacts, concrete elements such as mechanisms to use, devices that are available, etc. Consequently, a security pattern at domain specific level contains the respective information.

These two levels of abstractions are captured using new concepts related to the different kind of knowledge described by the pattern had; not with existing software constructs. In our work, we used the MDE philosophy. We do not use the software concepts (object or component constructs) recommended by Model-Driven Architecture (MDA), for example. However, the SEPM language is subject to target specific software modeling languages such as those recommended by MDA, using model transformation techniques. Regarding the well known MDA levels (PIM and PSM), there is an overlap between these levels and our two abstraction levels. For example, in the Intelligent Transport Systems (ITS) domain, the ISO/IEC 15118 highly recommends to use TLS for ensuring security properties. For that domain, we can find different platforms supporting such a DSPM pattern.

We also provide an accompanying formalization and validation framework to help precise specification of patterns based on the interactive Isabelle/HOL proof assistant. The resulting validation artifact's may mainly (1) complete the definitions, and (2) provide semantics for the interfaces and the properties in the context of S&D. Like this, validation artifacts may be added to the pattern for traceability concerns. In the same way, the domain refinement is applied during the formal validation process for the specification and validation of patterns.

Furthermore, we walk through a prototype of EMF tree-based editors supporting the approach. Currently the tool suite named *Semcomdt*[2] is provided as Eclipse plugins. The approach presented here has been evaluated on two case studies from the TERESA project[3] resulting in the development of a repository of S&D patterns with more than 30 S&D patterns.

The next step of this work consists in defining a correct-by-construction pattern-based security engineering process. It aims to provide the correct-by-construction integration of a design pattern into an application while offering a certain degree of liberty to the designer using it. In order to be able to validate the integration, we must have a formal specification of the pattern, i.e., its properties, constraints and related validation artifacts, as input to the pattern-based development process. Another objective for the near future is to provide automated tool support for pattern-based development, preferably based on a widely known and accepted model-based approach in industry such as UML [18]. For that, we plan to investigate the possibility to transform these design artifacts into UML [18] and their corresponding validation artifacts into OCL [19].

References

1. Alexander, C., Ishikawa, S., Silverstein, M.: A Pattern Language. Center for Environmental Structure Series, vol. 2. Oxford University Press, New York (1977)
2. Cheng, B., Cheng, B.H.C., Konrad, S., Campbell, L.A., Wassermann, R.: Using security patterns to model and analyze security. In: IEEE Workshop on Requirements for High Assurance Systems, pp. 13–22 (2003)
3. Douglass, B.P.: Real-time UML: Developing Efficient Objects for Embedded Systems. Addison-Wesley (1998)

[2] `http://www.semcomdt.org`

[3] `http://www.teresa-project.org/`

4. Fernandez, E.B., Yoshioka, N., Washizaki, H., Jürjens, J., VanHilst, M., Pernul, G.: Software Engineering for Secure Systems: Industrial and Research Perspectives. In: Mouratidis, H. (ed.) IGI Global, pp. 16–31 (2010)
5. Fuchs, A., Gürgens, S., Rudolph, C.: A Formal Notion of Trust – Enabling Reasoning about Security Properties. In: Nishigaki, M., Jøsang, A., Murayama, Y., Marsh, S. (eds.) IFIPTM 2010. IFIP AICT, vol. 321, pp. 200–215. Springer, Heidelberg (2010)
6. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
7. Gasparis, E., Nicholson, J., Eden, A.H.: LePUS3: An Object-Oriented Design Description Language. In: Stapleton, G., Howse, J., Lee, J. (eds.) Diagrams 2008. LNCS (LNAI), vol. 5223, pp. 364–367. Springer, Heidelberg (2008)
8. Di Giacomo, V., et al.: Using Security and Dependability Patterns for Reaction Processes. In: International Workshop on Database and Expert Systems Applications, pp. 315–319. IEEE Computer Society (2008)
9. Le Guennec, A., Sunyé, G., Jézéquel, J.-M.: Precise modeling of design patterns. In: Evans, A., Caskurlu, B., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 482–496. Springer, Heidelberg (2000)
10. Halkidis, S.T., Chatzigeorgiou, A., Stephanides, G.: A qualitative analysis of software security patterns. Computers & Security 25(5), 379–392 (2006)
11. Hamid, B., Gürgens, S., Jouvray, C., Desnos, N.: Enforcing S&D Pattern Design in RCES with Modeling and Formal Approaches. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 319–333. Springer, Heidelberg (2011)
12. Hatebur, D., Heisel, M., Schmidt, H.: A security engineering process based on patterns. In: Proceedings of the 18th International Conference on Database and Expert Systems Applications, DEXA 2007, pp. 734–738. IEEE Computer Society, Washington, DC (2007)
13. Jarvinen, H.M., Kurki-Suonio, R.: DisCo specification language: marriage of actions and objects. In: 11th International Conference on Distributed Computing Systems, pp. 142–151. IEEE Press (1991)
14. Jürjens, J.: UMLsec: Extending UML for Secure Systems Development. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 412–425. Springer, Heidelberg (2002)
15. Kim, D.K., France, R., Ghosh, S., Song, E.: A UML-Based Metamodeling Language to Specify Design Patterns. In: Patterns, Proc. Workshop Software Model Eng (WiSME) with Unified Modeling Language Conf. 2004, pp. 1–9 (2004)
16. Mikkonen, T.E.: Formalizing design patterns. In: Proceeding ICSE 1998 Proceedings of the 20th International Conference on Software Engineering. IEEE Press (1998)
17. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
18. OMG. OMG Unified Modeling Language (OMG UML), Superstructure (February 2009), `http://www.omg.org/spec/UML/2.2/Superstructure`
19. OMG. OCL 2.2 Specification (February 2010)
20. Schmidt, D.: Model-Driven Engineering. IEEE Computer 39(2), 41–47 (2006)
21. Schumacher, M.: Security Engineering with Patterns. LNCS, vol. 2754. Springer, Heidelberg (2003)
22. Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., Sommerlad, P.: Security Patterns: Integrating Security and Systems Engineering. Wiley Software Patterns Series. John Wiley & Sons (2006)
23. Yoshioka, N., Washizaki, H., Maruyama, K.: A survey of security patterns. Progress in Informatics (5), 35–47 (2008)

24. Zdun, U., Avgeriou, P.: Modeling Architectural Patterns Using Architectural Primitives. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, pp. 133–146. ACM, New York (2005)
25. Zhang, T., Jouault, F., Bezivin, J., Zhao, J.: A MDE Based Approach for Bridging Formal Models. In: Sixth International Symposium on Theoretical Aspects of Software Engineering, pp. 113–116 (2008)
26. Zurawski, R.: Embedded Systems. In: Embedded Systems Handbook. CRC Press Inc. (2005)

A. Isabelle/HOL Formalization and Validation of the Secure Communication Pattern

The complete formalisation and verification of the secure communication pattern, using Isabelle/HoL, are available online via `http://www.semcomdt.org/semco/resources/SCP_Isabelle.zip`.

A.1. DIPM Validation

```
theory DIPM imports Main
begin
(* Secure Communication DIPM Pattern definitions *)
...
definition authIfTrust :: "action ==> action ==> agent ==> bool" where
"authIfTrust a1 a2 a == trust a (precede a1 a2) -> auth a1 a2 a"

(* Secure Communication DIPM Pattern validation *)
theorem authIfTrustSendReceive : "authIfTrust Send Receive server"
apply (simp only : authIfTrust_def)
apply (simp only : trust_def)
apply (simp only : server_def)
apply (simp only : createProperty_def createAgent_def)
apply (simp only : Send_def Receive_def)
apply (simp only : createAction_def)
apply (simp only : auth_def)
apply (simp)
done
end
```

A.2. DSPM Formalization and Validation

```
theory DSPM imports Main
begin
(* Secure Communication DSPM Pattern definitions *)
...
definition trustRandom :: "agent==>propertyRandom==>bool" where
"trustRandom a p == p IN (propertiesRandom a)"
definition trustKey :: "agent ==> propertyKey==> bool" where
"trustKey a p == p IN (propertiesKey a)"
definition trustInConfidentiality :: "agent==> bool" where
"trustInConfidentiality a == (trustRandom a notPrecedeGenRandGenRand) AND
(trustKey a confPrivKeyS) AND (trustKey a confPrivKeyCA)"
definition trustInSignature :: "agent==> bool" where
"trustInSignature a ==  (trustMac a (precede genMac verifyMac))"
definition trustSCP :: "agent ==> bool" where
"trustSCP a == trustInConfidentiality a -> trustInSignature a"

(* Secure Communication DSPM Pattern validation *)
theorem proofTrustSCP : "trustSCP server"
apply (simp only : trustSCP_def)
apply (simp only : trustInConfidentiality_def trustInSignature_def)
apply (simp only : notPrecedeGenRandGenRand_def confPrivKeyS_def confPrivKeyCA_def)
```

```
apply (simp only : genMac_def verifyMac_def)
apply (simp only : notPrecede_def precede_def conf_def)
apply (simp only : createPropertyMac_def createActionMac_def)
apply (simp only : createPropertyRandom_def createActionRandom_def)
apply (simp only : createPropertyKey_def createActionKey_def)
apply (simp only : trustMac_def trustRandom_def trustKey_def)
apply (simp only : genRand_def privKey1_def privKey2_def)
apply (simp only : server_def)
apply (simp only : createActionMac_def  createActionRandom_def
createActionKey_def createAgent_def)
apply (simp)
apply (simp only : notPrecedeGenRandGenRand_def confPrivKeyS_def confPrivKeyCA_def)
apply (simp only : precedeGenMacVerifyMac_def)
apply (simp only : precede_def notPrecede_def conf_def)
apply (simp only : createPropertyMac_def  createPropertyRandom_def createPropertyKey_def)
apply (simp only : genMac_def verifyMac_def privKeyS_def privKeyCA_def genRand_def)
apply (simp only : createActionMac_def createActionRandom_def createActionKey_def)
apply (simp)
done
end
```

A.3. Correspondence between DIPM and DSPM Formalization and Validation

```
theory DIPMtoDSPM imports DIPM DSPM
begin
(* From DIPSM Pattern to DSPM Pattern definitions*)
...
definition buildAndApplyH :: "DSPM.agent ==> DIPM.agent ==>
(DSPM.action * DIPM.action) list ==> bool" where
"buildAndApplyH a1 a2 l == DSPM.trust a1 (buildProperty l) ->
DIPM.trust a2 (applyH l)"

(* From DIPSM Pattern to DSPM Pattern validation*)
theorem trustWithH : "buildAndApplyH DSPM.server DIPM.server h"
apply (simp only : buildAndApplyH_def)
apply (simp only : DIPM.trust_def DSPM.trust_def)
apply (simp only : buildProperty_def)
apply (simp only : applyH_def)
apply (simp)
apply (simp only : h_def)
apply (simp only : Send_def Receive_def)
apply (simp only : DIPM.createAction_def)
apply (simp only : genMac_def verifyMac_def)
apply (simp only : DSPM.createAction_def)
apply (simp only : DIPM.server_def DSPM.server_def)
apply (simp)
apply (simp only : DIPM.createAgent_def DSPM.createAgent_def)
apply (simp only : Send_def Receive_def genMac_def verifyMac_def)
apply (simp)
apply (simp only : precedeSendReceive_def precedeGenMacVerifyMac_def)
apply (simp only : DIPM.precede_def DIPM.createProperty_def DSPM.precede_def
DSPM.createProperty_def)
apply (simp)
apply (simp only : Send_def Receive_def genMac_def verifyMac_def)
apply (simp only : DIPM.createAction_def DSPM.createAction_def)
apply (simp)
done
end
```